Chapter 4

C What I/O

In This Chapter

- Reading the keyboard
- Understanding `printf()`
- Creating formatted output
- Understanding `scanf()`

Computers are all about input and output — the old I/O of days gone by, what the pioneers once sang about. The wimminfolk would want to dance real slow. Maybe cry. It was a sentimental thing, y'all — something that fancy, dooded-up city slickers read about in dime magazines.

A-hem!

Input and output: You type something in and get a response, ask a question and get an answer, put in two dollars in coins and get your soda pop — things along those lines. This goes along with what I present in Chapter 3: It is your job as a programmer to write a program that does something. At this point in the learning process, triviality is okay. Soon, however, you begin to write programs that really do something.

Introduce Yourself to Mr. Computer

To meet the needs of input and output — the old I/O — you can try the following program, WHORU.C — which is "who are you" minus a few letters. Please don't go calling this program "horror-you" (which could be spelled another way, but this is a family book).

The purpose of this program is to type your name at the keyboard and then have the computer display your name on the screen, along with a nice, friendly greeting:

```
#include <stdio.h>

int main()
{
   char me[20];

   printf("What is your name?");
   scanf("%s",&me);
   printf("Darn glad to meet you, %s!\n",me);

   return(0);
}
```

Type the preceding source code into your editor. Double-check everything. Don't bother with any details just yet. Type and hum, if it pleases you.

Save the file to disk. Name it WHORU.C.

Don't compile this program just yet. That happens in the next section.

- The `char me[20];` thing is a *variable declaration*. It provides storage for the information you enter (the *I* in I/O). You find out more about variables in Chapter 8.
- The new function here is `scanf()`, which is used to read input from the keyboard and store it in the computer's memory.
- Left paren is the `(` character. Right paren is the `)` character. *Paren* is short for parenthesis or a type of steak sauce. (It's also not a "real" word and is frowned on by English teachers of the high-and-tight bun.)

Compiling WHORU.C

Compile the WHORU.C source code. If you see syntax or other errors, double-check your source code with what is listed in this book. Ensure that you entered everything properly. Be on the lookout for jots and tittles — parentheses, double quotes, backslashes, percent signs, sneeze splotches, or other unusual things on your monitor's screen.

If you need to fix any errors, do so now. Otherwise, keep reading in the next section.

- Refer to Chapter 2 for more information on fixing errors and recompiling.
- A common beginner error: Unmatched double quotes! Make sure that you always use a set of "s (double quotes). If you miss one, you get an error. Also make sure that the parentheses and curly braces are included in pairs; left one first, right one second.

The reward

Enough waiting! Run the WHORU program now. Type **whoru** or **./whoru** at the command prompt and press the Enter key. The output looks like this:

```
What is your name?
```

The program is now waiting for you to type your name. Go ahead: Type your name! Press Enter.

If you typed **Buster**, the next line is displayed:

```
Darn glad to meet you, Buster!
```

- If the output looks different or the program doesn't work right or generates an error, review your source code again. Reedit to fix any errors and then recompile.
- I/O is input/output, what computers do best.
- I/O, I/O, it's off to code I go. . . .
- This program is an example that takes input and generates output. It doesn't do anything with the input other than display it, but it does qualify for I/O.

The WHORU.C source code mixes two powerful C language functions to get input and provide output: `printf()` and `scanf()`. The rest of this chapter tells more about these common and useful functions in detail.

More on `printf()`

The `printf()` function is used in the C programming language to display information on the screen. It's the all-purpose "Hey, I want to tell the user something" display-text command. It's the universal electric crayon for the C language's scribbling muscles.

The format for using the basic `printf` function is

```
printf("text");
```

`printf` is always written in lowercase. It's a must. It's followed by parentheses, which contain a quoted string of text, *text* (see the example). It's `printf()`'s job to display that text on the screen.

In the C language, `printf()`is a complete statement. A semicolon always follows the last parenthesis. (Okay, you may see an exception, but it's not worth fussing over at this point in the game.)

- Although *text* is enclosed in double quotes, they aren't part of the message that `printf()` puts up on the screen.
- You have to follow special rules about the text you can display, all of which are covered in Chapter 24.
- The format shown in the preceding example is simplified. A more advanced format for `printf()` appears later in this chapter.

Printing funky text

Ladies and gentlemen, I give you the following:

```
Ta da!  I am a text string.
```

It's a simple collection of text, numbers, letters, and other characters — but it's not a string of text. Nope. For those characters to be considered as a unit, they must be neatly enclosed in double quotes:

```
"Ta da!  I am a text string."
```

Now you have a string of text, but that's still nothing unless the computer can manipulate it. For manipulation, you need to wrap up the string in the bun-like parentheses:

```
("Ta da!  I am a text string.")
```

Furthermore, you need an engine — a *function* — to manipulate the string. Put `printf` on one side and a semicolon on the other:

```
printf("Ta da!  I am a text string.");
```

And, you have a hot dog of a C command to display the simple collection of text, numbers, letters, and other characters on the screen. Neat and tidy.

Consider this rabble:

```
He said, "Ta da! I am a text string."
```

Is this criminal or what? It's still a text string, but it contains the double-quote characters. Can you make that text a string by adding even more double quotes?

```
"He said, "Ta da! I am a text string.""
```

Now there are four double quotes in all. That means eight tick marks hovering over this string's head. How can it morally cope with that?

```
""Damocles" if I know."
```

The C compiler never punishes you for "testing" anything. There is no large room in a hollowed-out mountain in the Rockies where a little man sits in a chair looking at millions of video screens, one of which contains your PC's output, and, no, the little man doesn't snicker evilly whenever you get an error. Errors are safe! So why not experiment?

Please enter the following source code, DBLQUOTE.C. The resulting program is another "`printf()` displays something" example. But this time, what's displayed contains a double quote. Can you do that? This source code is your experiment for the day:

```
#include <stdio.h>

int main()
{
   printf("He said, "Ta da! I am a text string."");
   return(0);
}
```

Type the source code exactly as it appears, including the double quotes — four in all. (You notice right away that something is wrong if your editor color-codes quoted text. But work with me here.)

Save the source code file to disk as DBLQUOTE.C.

Compile and run the preceding program — if you can. Chances are that you encounter one of these errors instead:

```
dblequote.c: In function 'main':
dblequote.c:5: parse error before "Ta"
```

or

```
dblequote.c: In function 'main':
dblequote.c:6: syntax error before "Ta"
```

The `printf()` function requires a text string enclosed in double quotes. Your compiler knows that. After the second double quote in the string was encountered (before the word *Ta*), the compiler expected something else — something other than "Ta." Therefore, an error was generated. Figure 4-1 illustrates this in a cute way.

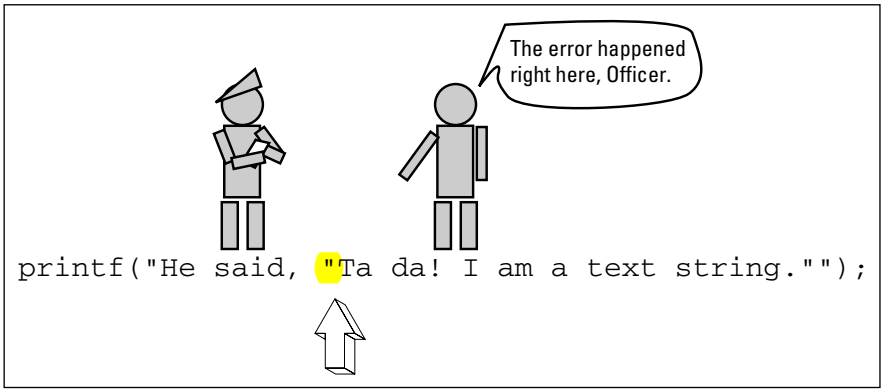

Figure 4-1: The C compiler detects something amiss in the `printf()` statement.

Obviously, there is a need to use the double-quote character in a string of text. The question is how to pass that character along to `printf()` without it ruining the rest of your day. The answer is to use an escape sequence.

In the olden days, programmers would have simply gone without certain characters. Rather than trip up a string with a double quote, they would have used two single quotes. Some ancient programmers who don't know about escape sequences still use these tricks.

Escape from `printf()`!

Escape sequences are designed to liven up an otherwise dull action picture with a few hard-cutting, loud-music moments of derring-do. In a programming language, *escape sequences* are used to sneak otherwise forbidden characters, or characters you cannot directly type at the keyboard, into text strings.

In the C language, escape sequences always begin with the backslash character (`\`). Locate this character on your keyboard now. It should be above the Enter key, though they often hide it elsewhere.

The backslash character signals the `printf()` function that an escape sequence is looming. When `printf()` sees the backslash, it thinks, "Omigosh, an escape sequence must be coming up," and it braces itself to accept an otherwise forbidden character.

To sneak in the double-quote character without getting `printf()` in a tizzy, you use the escape sequence `\"` (backslash, double quote). Behold, the new and improved Line 5:

```
printf("He said, \"Ta da! I am a text string.\"");
```

Notice the `\"` escape sequences in the text string. You see two of them, prefixing the two double quotes that appear in the string's midsection. The two outside double quotes, the ones that really are bookmarks to the entire string, remain intact. Looks weird, but it doesn't cause an error.

(If your text editor color-codes strings, you see how the escaped double quotes appear as special characters in the string, not as boundary markers, like the other double quotes.)

Edit your DBLQUOTE.C source code file. Make the escape-sequence modification to Line 5, as just shown. All you have to do is insert two backslash characters before the rogue double quotes: `\"`.

Save the changed source code file to disk, overwriting the original DBLQUOTE.C file with the newer version.

Compile and run. This time, it works and displays the following output:

```
He said, "Ta da! I am a text string."
```

- The `\"` escape sequence produces the double-quote character in the middle of a string.
- Another handy escape sequence you may have used in Chapter 1 is `\n`. That produces a "new line" in a string, just like pressing the Enter key. You cannot "type" the Enter key in a text string, so you must use the `\n` escape sequence.
- All escape sequences start with the backslash character.
- How do you stick a backslash character into a string? Use two of them: `\\` is the escape sequence that sticks a backslash character into a string.
- An escape sequence can appear anywhere in a text string: beginning, middle, or end and as many times as you want to use them. Essentially, the `\` thing is a shorthand notation for sticking forbidden characters into any string.
- Other escape sequences are listed in Chapter 24 (in Table 24-1).

The f means "formatted"

The function is called `printf()` for a reason. The *f* stands for *formatted*. The advantage of the `printf` function over other, similar display-this-or-that functions in C is that the output can be formatted.

Earlier in this chapter, I introduce the format for the basic `printf` function as

```
printf("text");
```

But the real format — *shhh!* — is

```
printf("format_string"[,var[,...]]);
```

What appears in the double quotes is really a *formatting string*. It's still text that appears in `printf()`'s output, but secretly inserted into the text are various *conversion characters*, or special "placeholders," that tell the `printf()` function how to format its output and do other powerful stuff.

After the format string comes a comma (still inside the parentheses) and then one or more items called *arguments*. The argument shown in the preceding example is *var*, which is short for *var*iable. You can use `printf()` to display the content or value of one or more variables. You do this by using special conversion characters in `format_string`. Figure 4-2 illustrates this concept rather beautifully.

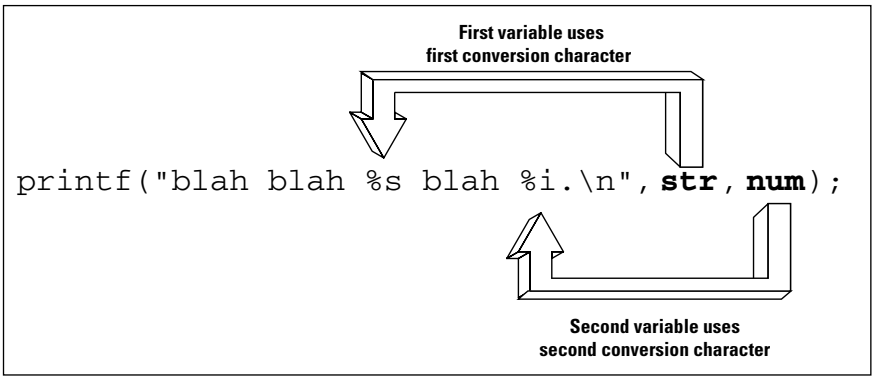

Figure 4-2: How `printf()` solves arguments.

The `[,...]` doohickey means that you can have any number of *var* items specified in a single `printf` function (before the final paren). Each *var* item, however, must have a corresponding placeholder (or conversion character) in `format_string`. They must match up, or else you get an error when the program is compiled.

A bit of justification

To demonstrate how `printf()` can format text, as well as use those handy *conversion characters* and *var* things I scared you with in the preceding section, how about a sample program?

For your consideration is the following source code, which I have named JUSTIFY.C. Marvel at it:

```
#include <stdio.h>

int main()
{
   printf("%15s","right\n");
   printf("%-15s","left\n");
   return(0);
}
```

What JUSTIFY.C does is to display two strings: `right` is right-justified, and `left` is left-justified. This makes more sense when you see the program's output rather than just look at the source code.

Enter this source code into your text editor.

In the first `printf` statement, the first string is `%15s` (percent sign, 15, little *s*). That's followed by a comma and then `right`, followed by the newline escape sequence, `\n` (backslash, little *n*).

The second `printf` statement is nearly the same thing, though with a minus sign before the `15` and the string `left` rather than `right`.

This program contains more C doodads than any other program introduced in the first three chapters in this book. Be careful with what you type! When you're certain that you have it right, save the file to disk as JUSTIFY.C.

Compile JUSTIFY.C. Fix any errors if you need to. Then run the program. Your output should look something like this:

```
          right
left
```

The word `right` is right-justified 15 spaces over; `left` is left-justified. This arrangement was dictated by the `%15s` formatting command in `printf()`. The `%15s` part of the formatting string didn't print at all. Instead, it *controlled* how the other string's contents appeared on the screen. That's the formatting power of `printf()` at work.

Maybe a little more help in understanding conversion characters

To drive home how `printf()` uses its formatting string and arguments, bring up the source code for the GOODBYE.C program into your text editor. Change Line 5 to read:

```
printf("%s","Goodbye, cruel
    world!\n");
```

`printf()` has been modified to contain a formatting string and an argument.

The formatting string is `%s`, which is the *string* (for *s*) placeholder.

The argument is a string of text: `"Goodbye, cruel world\!n"`.

Save the source code under a new filename, BYE.C. Compile and run. The output is the same as the original; you have merely used the `%s` in the `printf()` function to "format" the output.

Try this modification of Line 5:

```
printf("%s, %s
    %s\n","Goodbye","cruel",
    "world!");
```

Carefully edit Line 5 to look like what's shown in the preceding line. It has three string placeholders, `%s`, and three strings in double quotes (with commas between them). Save. Compile. Run. The output should be the same.

(If you get a compiling error, you probably have put a comma *inside* the double quotes, rather than between them.)

The JUSTIFY.C program shows you only a hint of what the `printf()` function can do. `printf()` can also format numbers in a remarkable number of ways, which is a little overwhelming to present right now in this chapter.

- In the `printf()` function, the first item in quotes is a formatting string, though it can also contain text to be displayed right on the screen.
- The percent character holds special meaning to `printf()`. It identifies a *conversion character* — what I call a "placeholder" — that tells `printf` how to format its output.
- The conversion character `s` means string: `%s`.
- Any numbers between the `%` and the `s` are used to set the *width* of the text string displayed. So, `%15s` means to display a string of text using 15 characters. A minus sign before the `15` means to left-justify the string's output.
- Doesn't "left-justify" sound like a word processing term? Yup! It's formatting!
- `printf()` doesn't truncate or shorten strings longer than the width specified in the `%s` placeholder.

- All this conversion-character stuff can get complex. Rest assured that seldom does anyone memorize it. Often, advanced programmers have to consult their C language references and run some tests to see which formatting command does what. Most of the time, you aren't bothered with this stuff, so don't panic.

scanf Is Pronounced "Scan-Eff"

Output without input is like Desi without Lucy, yang without yin, Caesar salad without the garlic. It means that the seven dwarves would be singing "Oh, Oh, Oh" rather than "I/O, I/O." Besides — and this may be the most horrid aspect of all — without input, the computer just sits there and talks *at* you. That's just awful.

C has numerous tools for making the computer listen to you. A number of commands read input from the keyboard, from commands that scan for individual characters to the vaunted `scanf()` function, which is used to snatch a string of text from the keyboard and save it in the cuddly, warm paws of a string variable.

- `scanf()` is a function like `printf()`. Its purpose is to read text from the keyboard.
- Like the *f* in `printf()`, the *f* in `scanf()` means *formatted.* You can use `scanf()` to read a specifically formatted bit of text from the keyboard. In this chapter, however, you just use `scanf()` to read a line of text, nothing fancy.

Putting scanf together

To make `scanf()` work, you need two things. First, you need a storage place to hold the text you enter. Second, you need the `scanf` function itself.

The storage place is called a *string variable. String* means a string of characters — text. *Variable* means that the string isn't set — it can be whatever the user types. A string variable is a storage place for text in your programs. (Variables are discussed at length in Chapter 8.)

The second thing you need is `scanf()` itself. Its format is somewhat similar to the advanced, cryptic format for `printf()`, so there's no point in wasting any of your brain cells covering that here.

An example of using `scanf()` reads in someone's first name. First, you create a storage place for the first name:

```
char firstname[20];
```

This C language statement sets aside storage for a string of text — like creating a safe for a huge sum of money that you wish to have some day. And, just like the safe, the variable is "empty" when you create it; it doesn't contain anything until you put something there.

Here's how the preceding statement breaks down:

- `char` is a C language keyword that tells the compiler to create a character variable, something that holds text (as opposed to numbers).
- `firstname` is the name of the storage location. When the source code refers to the variable, it uses this name, `firstname`.
- `[20]` defines the size of the string as being able to hold as many as 20 characters. All told, you have set aside space to hold 20 characters and named that space — that *variable* — `firstname`.
- The semicolon ends the C language statement.

The next step is to use the `scanf()` function to read in text from the keyboard and store it in the variable that is created. Something like the following line would work:

```
scanf("%s",&firstname);
```

Here's how this statement works:

- `scanf()` is the function to read information from the keyboard.
- `%s` is the string placeholder; `scanf()` is looking for plain old text input from the keyboard. Pressing the Enter key ends input.
- The text input is stored in the string variable named `firstname`. The ampersand is required here to help `scanf()` find the location of the string variable in memory.
- The semicolon ends the C language statement.

Between the variable and `scanf()`, text is read from the keyboard and stored in the computer's memory for later use. The next section coughs up an example.

- If you're writing a C program that requires input, you must create a place to store it. For text input, that place is a string variable, which you create by using the `char` keyword.
- Variables are officially introduced in Chapter 8 in this book. For now, consider the string variable that `scanf()` uses as merely a storage chamber for text you type.
- The formatting codes used by `scanf()` are identical to those used by `printf()`. In real life, you use them mostly with `printf()` because there are better ways to read the keyboard than to use `scanf()`. Refer to Table 24-2 in Chapter 24 for a list of the formatting percent-sign placeholder codes.
- Forgetting to stick the `&` in front of `scanf()`'s variable is a common mistake. Not doing so leads to some wonderful *null pointer assignment* errors that you may relish in the years to come. As a weird quirk, however, the ampersand is optional when you're dealing with string variables. Go figure.

The miracle of `scanf()`

Consider the following pointless program, COLOR.C, which uses two string variables, `name` and `color`. It asks for your name and then your favorite color. The final `printf()` statement then displays what you enter.

```
#include <stdio.h>

int main()
{
   char name[20];
   char color[20];

   printf("What is your name?");
   scanf("%s",name);
   printf("What is your favorite color?");
   scanf("%s",color);
   printf("%s's favorite color is %s\n",name,color);
   return(0);
}
```

Enter this source code into your editor. Save this file to disk as COLOR.C. Compile.

If you get any errors, double-check your source code and reedit the file. A common mistake: forgetting that there are two commas in the final `printf()` statement.

Run the program! The output looks something like this:

```
What is your name?dan
What is your favorite color?brown
dan's favorite color is brown
```

In Windows XP, you have to run the command by using the following line:

```
.\color
```

The reason is that COLOR is a valid console command in Windows XP, used to change the foreground and background color of the console window.

Experimentation time!

Which is more important: the order of the `%s` doodads or the order of the variables — the arguments — in a `printf` statement? Give up? I'm not going to tell you the answer. You have to figure it out for yourself.

Make the following modification to Line 12 in the COLOR.C program:

```
printf("%s's favorite color is %s\n",color,name);
```

The order of the variables here is reversed: `color` comes first and then `name`. Save this change to disk and recompile. The program still runs, but the output is different because you changed the variable order. You may see something like this:

```
brown's favorite color is Dan.
```

See? Computers *are* stupid! The point here is that you must remember the order of the variables when you have more than one listed in a `printf()` function. The `%s` thingies? They're just fill-in-the-blanks.

How about making this change:

```
printf("%s's favorite color is %s\n",name,name);
```

This modification uses the `name` variable twice — perfectly allowable. All `printf()` needs are two string variables to match the two `%s` signs in its formatting string. Save this change and recompile. Run the program and examine the output:

```
Dan's favorite color is Dan
```

Okay, Lois — have you been drinking again? Make that mistake on an IRS form and you may spend years playing golf with former stockbrokers and congressmen. (Better learn to order your variables now.)

Finally, make the following modification:

```
printf("%s's favorite color is %s\n",name,"blue");
```

Rather than the `color` variable, a *string constant* is used. A *string constant* is simply a string enclosed in quotes. It doesn't change, unlike a variable, which can hold anything. (It isn't variable!)

Save the change to disk and recompile your efforts. The program still works, though no matter which color you enter, the computer always insists that it's "blue."

- The string constant `"blue"` works because `printf()`'s `%s` placeholder looks for a string of text. It doesn't matter whether the string is a variable or a "real" text string sitting there in double quotes. (Of course, the advantage to writing a program is that you can use variables to store input; using the constant is a little silly because the computer already knows what it's going to print. I mean, ladies and gentlemen, where is the I/O?)
- The `%s` placeholder in a `printf()` function looks for a corresponding string variable and plugs it in to the text that is displayed.
- You need one string variable in the `printf()` function for each `%s` that appears in `printf()`'s formatting string. If the variable is missing, a syntax boo-boo is generated by the compiler.
- In addition to string variables, you can use string constants, often called *literal* strings. That's kind of dumb, though, because there's no point in wasting time with `%s` if you already know what you're going to display. (I have to demonstrate it here, however, or else I have to go to C Teacher's Prison in Connecticut.)

- Make sure that you get the order of your variables correct. This advice is especially important when you use both numeric and string variables in `printf`.
- The percent sign (%) is a holy character. *Om!* If you want a percent sign (%) to appear in `printf`'s output, use two of them: `%%`.